Object Frameworks for Agent System Development

Steven Y. Goldsmith
Shannon V. Spires
Laurence R. Phillips

Sandia National Laboratories
Albuquerque, NM 87115-5800
(505) 845-8926
sygolds@sandia.gov
lrphill@sandia.gov
svspire@sandia.gov

Abstract

In this paper we present and analyze an instance of a mutli-agent development system we call the *standard agent system* that enables the construction of distributed agent-based software systems for R&D, industrial, and government applications. The standard-agent system is implemented as an extensible object framework that provides a means for constructing and customizing agent objects through specialization of standard base classes and by composition of component classes. The framework has been used to develop three major applications and has supported numerous research projects. Our position on agent toolkits is that object-centered frameworks containing standard design patterns and design patterns specialized for the agent domain provide the best compromise between generality and flexibility. Moreover, frameworks (systems of design patterns) enable reuse, contribute to the reliability, maintainability, and extensibility of an agent application. Design patterns capture the conceptualizations of common agent attributes while enabling variations of a design to coexist in the same system. We discuss the use of design patterns applied to agent programming and describe a design pattern that captures a general goal-directed deliberation mechanism. The agent programming environment is supported by supported by advanced object services such as distributed objects, object data bases, lifecycle protocols, and interfaces to the World-Wide Web. We discuss the relevance of these services to agent development.

Introduction

In this paper we discuss an instance of a mutli-agent development system we call the *standard agent system* that enables the construction of distributed agent-based software systems for R&D, industrial, and government applications. The standard-agent system is implemented as an extensible object framework (Goldsmith 1997) that provides a means for constructing and customizing agent objects through specialization of standard base classes (an architecture-driven framework) and by composition of component classes (a data-driven framework)[1]. The framework has been used to conduct research on distributed agent systems and to develop three major applications: (1) A multi-agent electronic commerce system in the manufacturing and transportation domain (Goldsmith, Phillips, and Spires 1998); (2) a simulator for collective robotics research (Goldsmith and Robinett 1998); and (3) a multi-agent collaborative search engine for the World-Wide Web (Phillips, Spires, and Goldsmith 1998). These software agents tend to be complex. They contain both deliberative and reflexive components (Wooldridge and Jennings 1995), are knowledge-intensive, and are usually distributed within a network

[1] "A framework is a set of cooperating classes that make up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes a framework to a particular application by subclassing and composing instances of framework classes" (Gamma et al 1995).

environment. Two applications exhibit explicit collaboration through shared goals based on joint intentions (Cohen & Levesque 1991; Jennings 1995; Tambe 1997). The framework relies on a considerable software substrate that supports distributed objects, object lifecycle protocols, object data bases, and interfaces to the World-Wide Web.

Our experience with frameworks for agent development has lead us to the position that object-centered frameworks containing standard design patterns (Gamma, Helm, Johnson, Vlissides 1995) and patterns specialized for the agent domain provide the best compromise between generality and flexibility. Moreover, frameworks enable rapid development cycles without sacrificing quality by leveraging reusable designs and reusable objects. Since agent development is fundamentally an activity of software engineering, all the necessary engineering "ilities" must be addressed: reliability, maintainability, vulnerability, extensibility (addressed by the framework approach), safety, and others. Clearly an agent toolkit must at least foster good programming practices and encourage desirable non-functional engineering attributes in addition to simplifying the functional development of the agent application. Frameworks contribute positively to most of these attributes. We suggest that agent development toolkits provide a collection of design patterns in the form of a development framework for agents that supports the development of single agent and multi-agent systems: (1) Standard Agency; (2) Standard Agent; and (3) Standard Goal. Additional design patterns apply to component subsystems of the Standard Agent pattern. The remainder of this paper discusses the object development environment, the standard agent framework, and summarizes the relevance to general issues in agent toolkits.

Supporting Software

The agents we refer to in this paper are complex objects that must be supported by an advanced object programming environment. In our experience, agents system development for both research projects and fielded applications requires a sophisticated object programming substrate that supports the following services:

Distributed Objects - A distributed object system that supports proxies, copies, replicants, remote method invocation, global object identity, class coercion, and object brokering. A network agent uses distributed object services to exchange messages with other agents, to register itself in a foreign domain, and to provide automatic coercion of objects into proxies, replicants or copies.

Object Data Base - An object oriented data base, fully integrated with the distributed object system, that provides persistence, identity, storage of composites, large binary objects such as text and images, versioning, transactions, and meta-data storage. The object data base gives an agent multiple independent persistent object stores that can contain specific ontologies and can record state information for fault recovery.

Object Lifecycle Protocol - A protocol for management of the entire object lifecycle, fully integrated with the distributed object service and the object data base service, that includes an object factory, an object destruction protocol, examination and update protocols, object collections and iterators, composite objects, and protocols for handling unknown and incompletely specified composites. Agents use the object lifecycle protocol to create and destroy complex objects (including agents) while maintaining referential integrity. Agents can represent incomplete objects and use standard methods to reason about incomplete information. Agents can use standard protocols to handle object collections such as data base records, cases for case-based reasoning, and images.

Object-HTML- A protocol for rapid development of complex HTML forms interfaces that includes a compiler that renders an HTML stream as a composite object (tree) that can be searched and modified by an agent. This service enables agents to conduct complex elicitation sessions with users through a web browser.

Meta-Object Protocol - A protocol That enables introspection and intercession on class and method metaobjects. An agent can reason about classes and methods, and can define new classes and methods at runtime to alter its internal functions.

It can be argued that not every agent application requires these advanced object services. However, most full-blooded agent applications with some degree of mission criticality will require all of them. Since the Internet and World-Wide Web are currently the most fertile ground for fielding multiagent applications, these services become the point of departure for the development of network agent systems. An agent toolkit can be evaluated along a support *services* dimension by determining the degree to which its underlying development environment provides these services.

Agent Design Patterns

Design patterns are often partitioned into three kinds: conceptual patterns, design patterns, and programming patterns (Riehle and Zullighoven 1996). A conceptual pattern is described by terms and concepts from a particular application domain. A pattern describing the concept of joint intentions is an example of a conceptual pattern in the agent domain. A design pattern is described in terms of software constructs, such as objects, classes, inheritance, and operators. It elaborates a conceptual pattern by specifying a software implementation. An example of a design pattern for agent goals is given below. Programming patterns are expressed in programming language elements. Programming patterns commit a design pattern to a language-specific implementation. Macros are typically used to implement programming patterns.

The example design pattern shown below describes a mechanism for controlling the state of a user-specified goal. The pattern provides a standard protocol that the developer can use to alter the state of a goal according to a well-known state transition diagram. It captures concepts that are common in goal-directed reasoning, or deliberative agents, in terms of classes, objects, and operators. As such, it is language-independent and can be implemented in any object-oriented language. However, it does assume the existence of a meta-object function that changes the class of an instance at runtime.

Goal

Intent

Separate meta-reasoning about the state of a goal from the implementation of the reasoning mechanism used to satisfy the goal.

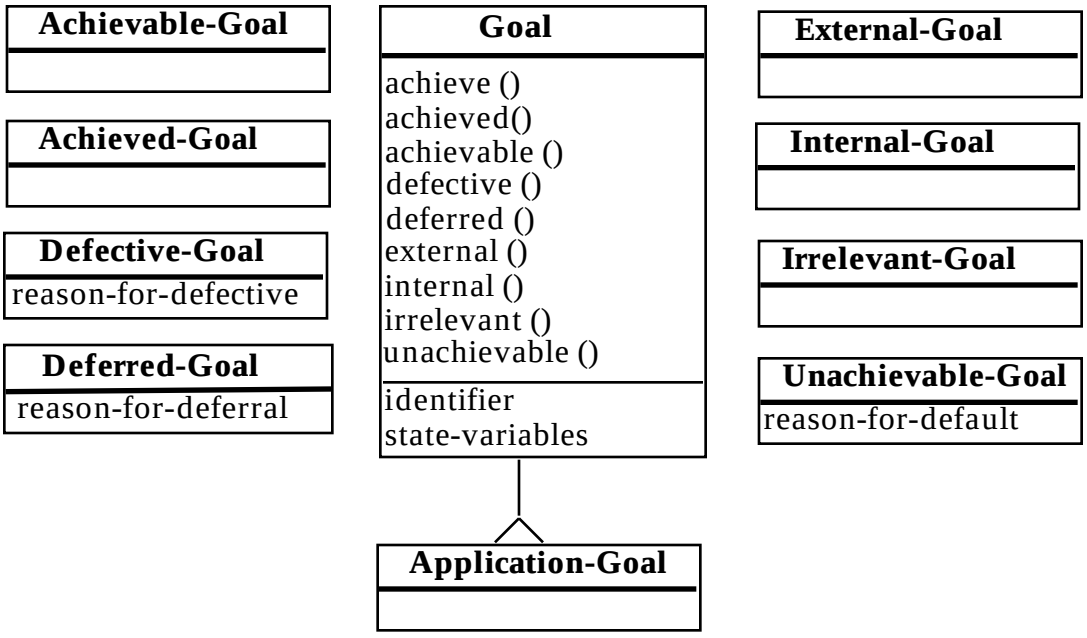
Motivation

The state of a goal can change over time. The deliberative mechanism can reason about the state of a goal independent of the specific reasoning mechanism used to attempt to achieve the goal. A set of standard goal states is implemented to provide the programmer with a meta-reasoning protocol that alters the state of a goal according to user-specified conditions. An achievable goal is a goal that the agent believes can be satisfied and the agent will expend computational resources to attempt its satisfaction. An achieved goal is a goal that an agent believes it has successfully satisfied. A defective goal is an ill-formed goal that lacks critical state data or is otherwise incoherent and cannot be achieved. A deferred goal is a goal that the agent believes cannot currently be satisfied without external information or a change in the state of the environment. External goals are goals that are copied for export to another agent or are goals received from another agent. An internal goal is a private copy of a goal owned by the agent. Irrelevant goals are goals that no longer have motivational support. Unachievable goals are well-formed goals that the agent believes cannot be satisfied.

Applicability

The Goal pattern is used to implement a general mechanism for handling goals that provides standard classes and semantics for goal states. The programmer can create application subclasses of the goal class and specialize the *achieve* method to implement a reasoning mechanism for the subclass. The state classes can also be subclassed to create a custom state-handling mechanism.

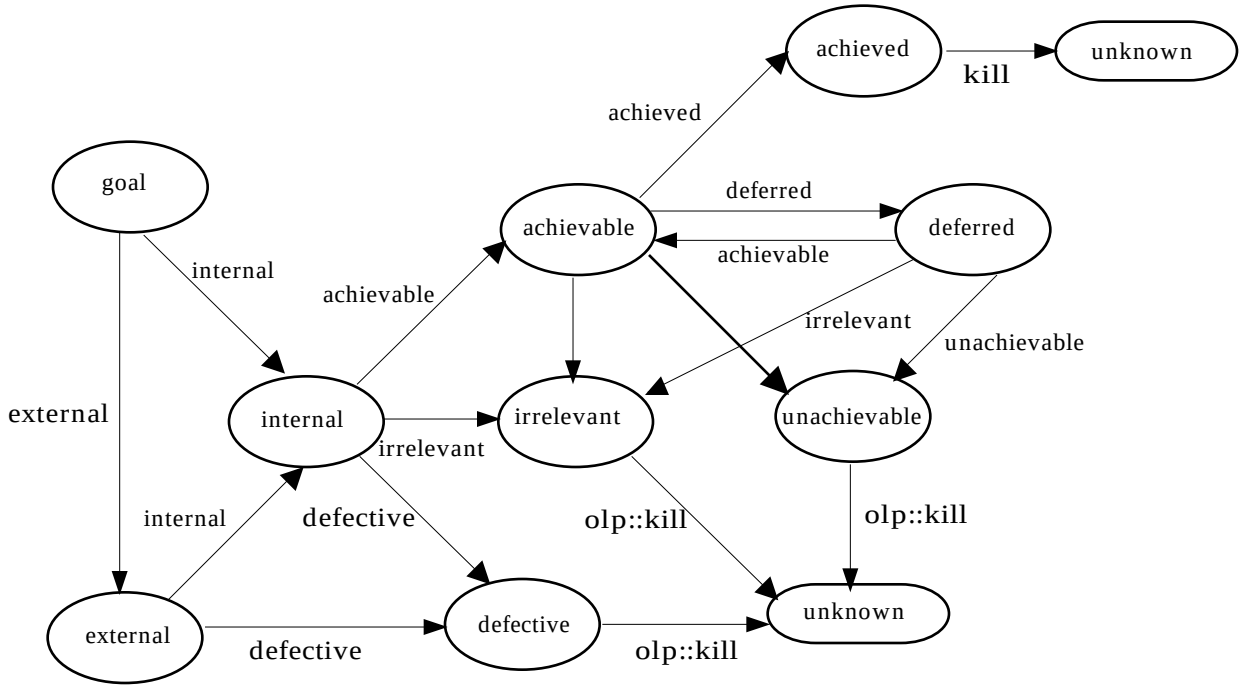
Structure



Participants

The state operators alter the class of the user goal to reflect the current state of the goal. Each operator pushes its respective state class on the precedence list of the user goal and changes the class of the goal to the state class. Side effects such as removing the goal from an agenda (irrelevant-goal, unachievable-goal, defective-

goal) or placing the goal on an agenda (achievable-goal, deferred-goal) can be implemented in the state operator methods. A goal
object can be destroyed (changed to the class olp::unknown) by the kill operation imported from the Object Lifecycle Protocol package. The state evolution of a goal is illustrated in the figure below. The state operator methods are used by the programmer to assert the state of the goal. Overloaded state operators enforce the admissible state transitions by performing a NOP. For example, the achieved operator is defined on the class goal to implement a NOP, and asserts the achieved state only on goals of class achievable.



Collaborations

The client invokes a state operator from a method specialized to handle the application goal. The defective, achieved, unachievable, irrelevant, and deferred methods are typically invoked in a specialized achieve method. The internal operator is typically invoked by the agent's communications interface or by the make-goal method.

Consequences

The goal pattern gives the programmer the freedom to choose the state variables and the reasoning mechanism used to achieve an application goal while providing a standard mechanism for handling the state of the goal.

Standard Agent Framework

The standard agent framework (SAF) is an example of an extensible object framework that supports numerous standard and custom design patterns. The SAF provides a means for constructing and customizing multiagent systems through specialization of standard base classes (architecture-driven framework) and by composition of component classes (data-driven framework). The standard agent framework comprises two associated abstract classes: *agent* and agency .[2] An agency identifies an independent locus of processes, activities, and knowledge typically associated with an company, organization, department, site, household, machine, or some other natural partitioning of the application domain. The underlying

[2] "An abstract class is a design for a single object. A framework is the design of a set of objects that collaborate to carry out a set of responsibilities. Thus, frameworks are larger scale designs than abstract classes." (Johnson and Russo 1991).

assumption is that the application is naturally modeled as a group of interacting agencies. The agency provides a containing context for a collection of agents. The activities of the agency are conducted by its constituent agents. Agents inhabit an agency for the express purpose of providing *services* , including intra-agency communications, that maintain the functioning of the agency and lead to satisfaction of the ultimate objectives of the agency. An agent performs domain-specific tasks on behalf of human actors and other agents. The class agent can be subclasses and the methods that implement primitive agent behaviors can be specialized to achieve agent *speciation*. Agent species are differentiated at an intrinsic level in terms of their normative goals and behaviors, largely determined by composition through instantiation with behavioral components. Different species of agents will vary along one or more dimensions that are captured in the object protocols that implement the particular design decision. Standardized agent systems are implemented by the specialization and instantiation of four concrete classes: (1) Standard Agency; (2) Standard Agent; (3) Human Actor; and (4) Resources. The class Standard Agency is an elaboration of the agency concept that includes human activities within the agency and devices for data-oriented activities such as storage and communications. An instance of Standard Agency is a persistent, identity-bearing composite object that contains collections of the component classes Standard Agent, Human Actor, and Resources. A standard agency object inherits collection behavior from the collection mixin class, and uses the standard collection iterator pattern (Gamma et al 1995) to provide an enumeration method for its components. The protocols for the Standard Agency class are primarily introspective methods that return component instances to enable an overview of agency status for debug and display purposes. The state of the agency object evolves in time as human actors, agents, and resources are added and deleted. The agency object provides only a containing context for its components, but conforms to the Abstract Factory design pattern (Gamma et al 1995) that implements a complex construction protocol for agencies.

The class Standard Agent implements instances of agents that have specific attributes: autonomy, social ability, reflexivity, and pro-activeness. An instance of Standard Agent is a composite object containing collections of component objects that define its structure and behavior. The primary Standard Agent protocols are an interface mechanism that enables interaction with other agents and human actors, a reflexive action mechanism for rapidly responding to event objects in its environment, and a generic inference mechanism for achieving explicit goals. The interaction and inference protocols can be specialized with methods that violate the standard design pattern and implement other agent architectures and mechanisms. Agents are self-contained threads of execution that execute both periodically and through immediate scheduling. The concrete component class Standard Goal implements a general achievement goal. An agent's agenda contains descendants of Standard Goal that define the agent's motivations. Standard Goal may also be subclasses and its protocols specialized to implement modular knowledge to achieve well-known classes of goals. Mixin goal behaviors are possible that define reusable canonical goal behaviors useful in many applications. For example, conducting an interactive session with a human informant has many common aspects to it that are independent of the domain or subject. These common aspects are captured in the mixin elicitation-goal that abstracts away from the subject goal design things like user timeout handling, stream and i/o management, and human-agent synchronization. A specialized concrete subclass of standard goal implements a Joint Persistent Goal (JPG) in the design pattern supporting collaboration through joint intentions. Then joint intentions pattern supports extensions for team formation (Tambe 1997) as well.

Human actors are objects in the agent domain that represent people interacting with agents through an interface device. A person logs into the agency temporarily for a work session through an interface device. Human actor objects are temporary objects that contain an

interface address, an interface object that captures the display, data entry and control functions currently available to the person, and a persistent person object that holds personal data, passwords, email address, and an account object that provides access to past and current workspaces. A workspace object contains objects created and stored by the person during work sessions.

Agents and human actors have access to resources such as databases, fax machines, telephones, email handlers, signal processors, controllers, and other useful services. Resource objects provide concurrency control and access protocols for agency resources. Subclasses of the resource class implement objects representing data bases, fax machines, printers, email ports, EDI ports and other commonplace legacy devices in the agency environment. Interfaces to these devices are abstracted in numerous standard design patterns.

The Standard Agent Framework supports distributed agent systems through the distributed objects layer. Agency objects may be distributed in a network environment to create a collaborative enterprise structure of interconnected agencies. Agency objects are represented The fundamental activity conducted among distributed agencies is the trading of domain objects through proxy agents that represent one agency within the agent collection of another agency. These proxy objects delegate all messages (except for a local request for identifying information about the represented agency) to the actual agent residing in the agency. Public proxies are registered in an agency network phonebook with a well-know address. To find other agencies, an agent issues one or more queries to the phonebook and is returned the proxy objects matching the query. The agent proxies interned within an agency form a persistent network of agencies. Such networks are called durable *proxy networks*. Agents can also directly intern proxies to other agents to form private networks of collaborating agents.

Agents can communicate via variety of communications standards. A subclass of Standard Goal called KQML-Goal implements sensing, receiving, and interpreting KQML messages among agents. The JPG goal makes use of KQML-Goal as a subgoal to implement cooperation via shared goals. Agents can also communicate directly with WWW servers via a standard object representation of HTML pages and a protocol that provides a stream abstraction for the server in the agent environment as a resource object.

Discussion

We have exhibited an example of an agent design pattern and discussed an example of an agent development framework. We suggest that agent toolkits can be evaluated according to the support services they provide. Design patterns, organized into a development framework, provide a means to evaluate agent development toolkits. Agent development systems can be evaluated against a collection of standard design patterns, developed by the community, that describe common architectures such as Belief-Desire-Intention (Georgeff & Rao 1995). Design patterns capture agent components in standard object notation that can be implemented in a variety of languages. The pattern description format accepted in the design pattern research enables clear communication of the specifications for the pattern and enables fair comparison of toolkits.

References

Cohen, P., and Levesque, H. 1991. Teamwork. *Nous, 25.*

Gamma, E.,Helm, R., Johnson,R., Vlissides,J., . *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley 1995. ISBN 0-201-63361-2.

Georgeff, M. and Rao, A. 1995. BDI agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95*).

Goldsmith, S. Y. 1997. *The Standard Agent Framework.* Advanced Information Systems Laboratory Report, Sandia National Laboratories, Albuquerque, NM.

Goldsmith, S. Y., Phillips, L. R., Spires, S. V. 1998. A Collaborative Multiagent System for International Shipping. Accepted for the Workshop on Agent-Mediated Trading, Agents '98 Conference, Minneapolis MN.

Goldsmith, S. , Robinett, R. 1998. Collective Search by Mobile Robots using Alpha-Beta Coordination. Accepted for the Workshop on Collective Robotics (CRW98), Agent World 98, Paris, France.

Phillips, L. R., Spires, S. V., Goldsmith, S. Y. 1998. Distributed Search by an Agent Collective. Submitted to the Workshop on Learning for Text Categorization, AAAI-98, Madison WI.

Jennings, N. 1995. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence, 75.*

Johnson, R., Russo, V. 1991. In *Reusing Object-Oriented Designs.*

Riehle D., Züllighoven H. 1996. Understanding and using patterns in software development. In *Theory and Practice of Object Systems 2, 1*.

Tambe, M. 1997. Towards Flexible Teamwork. *Journal of Artificial Intelligence Research 7*, 83-124.

Woolridge, M. and Jennings, N. 1995. Intelligent Agents: Theory and Practice. In Knowledge *Engineering Review*.